ILLINOIS INSTITUTE OF TECHNOLOGY

Big Data Technologies - CSP 554

Project Report
on
Comparative Analysis of Cloud-Based NoSQL Databases

Team Members

Urjita Saxena	**A20578349**
Harsh Sharma	**A20563703**
Nikita Sharma	**A20588827**
Trushaliben Chandulal Tanti	**A20598441**

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF COMPUTING

1. Overview

As applications increasingly demand real-time, large-scale data processing, distributed NoSQL databases have become vital for scalability and flexibility. Unlike traditional relational databases, NoSQL systems provide horizontal scaling, schema flexibility, and fault tolerance.

This project compares AWS DynamoDB, MongoDB Atlas, and Apache Cassandra to evaluate how their architectures influence performance, scalability, and cost efficiency. Each represents a distinct data model (key-value/document, document, and wide-column) and consistency approach, allowing an empirical study of CAP theorem trade-offs in practice.

Traditional relational databases often struggle to handle the scalability, flexibility, and high-throughput demands of modern applications. NoSQL databases address these challenges by efficiently managing large volumes of unstructured or semi-structured data.

However, NoSQL systems differ significantly in design. MongoDB (document store), Cassandra (wide-column store), and DynamoDB (key-value store) each balance consistency, latency, and fault tolerance differently. This project provides an empirical comparison of these databases to highlight their strengths, limitations, and most suitable use cases across varied cloud workloads.

2. Review of the Literature

The explosion of big data applications has made scalable, schema-flexible, and high-throughput databases essential. Cloud-based NoSQL systems—specifically Amazon DynamoDB (key-value/document store), MongoDB Atlas (document store), and Apache Cassandra/DataStax Astra DB (wide-column store)—represent three distinct architectural approaches and CAP theorem trade-offs. This literature review compares them across architecture, data models, performance (individual and bulk operations), scalability, consistency, security, cost, benefits, and drawbacks, incorporating recent sources and benchmarks (2024-2025).

The three databases selected for in-depth evaluation in this project are:

• **Amazon DynamoDB** (fully managed key-value/document store, AWS):

Amazon DynamoDB is a fully managed NoSQL database provided by AWS that combines key-value and document storage models. It achieves predictable single-digit millisecond latency at any scale through consistent hashing, automatic data partitioning, and on-demand or provisioned capacity modes. By default, it favors availability and partition tolerance (AP in the CAP theorem) with eventual consistency, while offering optional strong consistency for specific operations at the cost of slightly higher latency and reduced throughput. Its serverless nature, global tables for multi-region replication,

and adaptive capacity features make it particularly suitable for applications requiring minimal operational overhead and seamless scaling (Amazon Web Services, 2025).

• **Apache Cassandra / DataStax Astra DB** (distributed wide-column store):

Apache Cassandra is an open-source, decentralized wide-column store originally designed by Facebook and inspired by both Dynamo and Bigtable. It uses a peer-to-peer ring architecture with no single point of failure, gossip protocols for cluster metadata, and tunable consistency levels that allow developers to balance availability and consistency per operation. Data is stored in log-structured merge trees with periodic compaction, making Cassandra exceptionally efficient for high-write-throughput workloads such as time-series data, logging, and IoT applications. Recent versions (including Cassandra 5.0 and DataStax Astra DB enhancements) have introduced improved storage engines, zero-downtime topology changes, and better observability, reinforcing its position as a robust choice for globally distributed, write-intensive systems (Apache Software Foundation, 2025; DataStax, 2024).

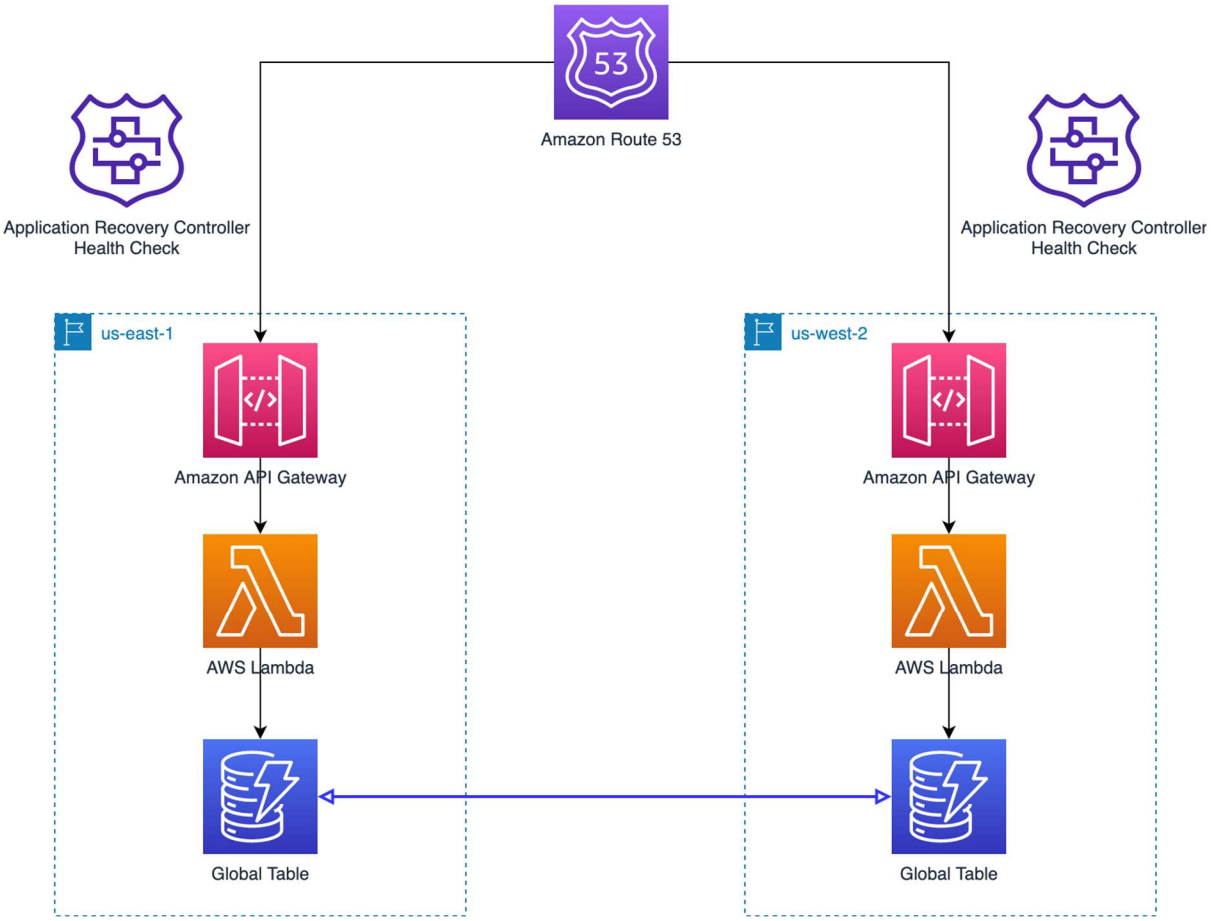
• **MongoDB Atlas** (managed document store with sharding and replica sets):

MongoDB Atlas is the fully managed cloud version of MongoDB, a document-oriented database that stores data in flexible, JSON-like BSON documents. It provides strong consistency by default within replica sets, supports multi-document ACID transactions since version 4.0, and scales horizontally via automated sharding. Its rich query language, secondary indexes, aggregation framework, and recent additions such as time-series collections, columnar indexes, and vector search make it particularly developer-friendly and versatile for applications requiring complex queries and rapid iteration. Although it introduces more operational complexity than fully serverless alternatives, its balance of flexibility, consistency, and querying power has made it one of the most widely adopted NoSQL solutions (MongoDB Inc., 2025).

3. Architecture and Data Models

3.1.DynamoDB:

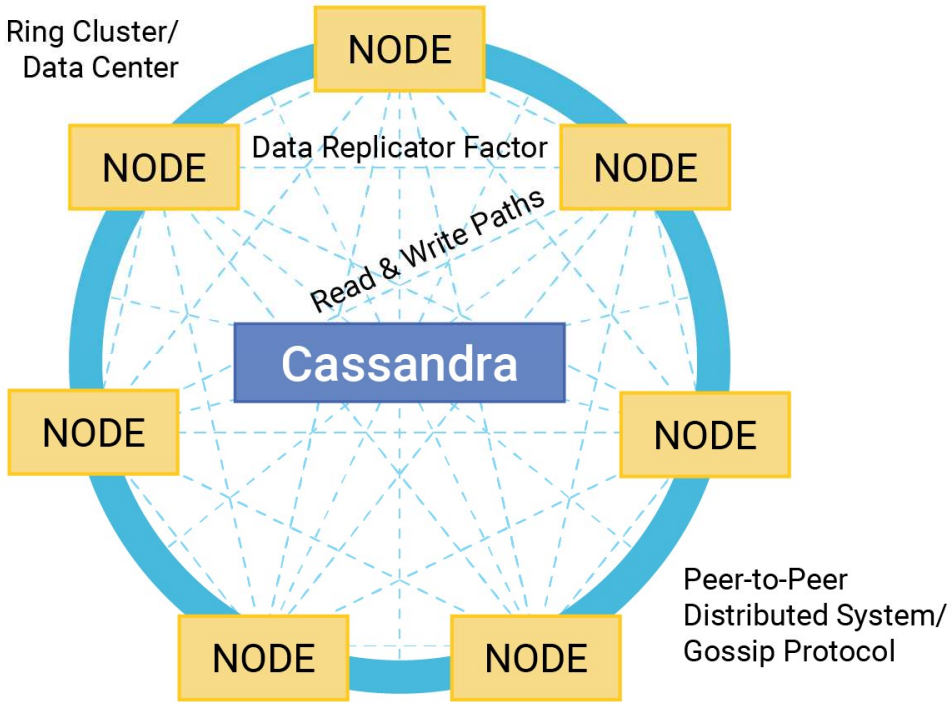
DynamoDB uses a serverless, ring-based architecture with automatic partitioning via partition keys (and optional sort keys) and built-in multi-AZ replication. It supports both key-value and document (JSON) models up to 400 KB per item.



This diagram shows a classic multi-region, serverless setup built around DynamoDB Global Tables for achieving low-latency reads/writes and disaster recovery. Client requests first hit Route 53, which uses health checks and latency-based routing (or failover policies via Application Recovery Controller) to send traffic to the nearest healthy region (here us-east-1 or us-west-2). In each region, an API Gateway endpoint receives the request, triggers a Lambda function for business logic, and the Lambda directly reads from or writes to the local replica of the Global Table. Because Global Tables automatically replicate data asynchronously across regions with eventual consistency (usually within a second), both regions always have near-identical data while staying fully available even if one region goes down completely. The whole stack is serverless, auto-scaling, and requires zero infrastructure management, making it a go-to pattern when you need sub-10 ms latency worldwide plus strong regional fault tolerance (Amazon Web Services, 2025).

Cassandra DB:

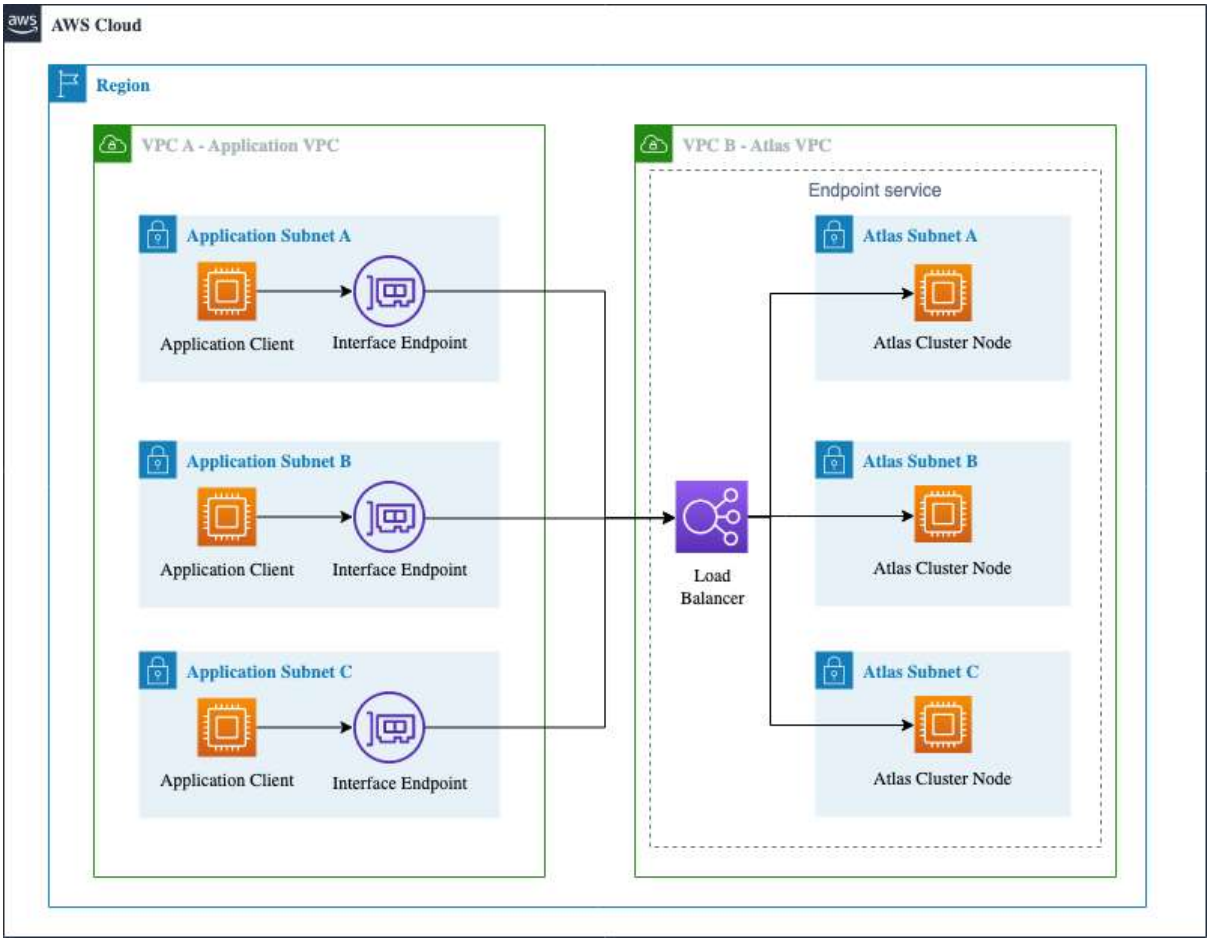
Cassandra/Astra DB follows a decentralized masterless ring topology (consistent hashing with vnodes, gossip protocol), using a wide-column (column-family) model optimized for massive write-heavy workloads.



This diagram captures the essence of Apache Cassandra's decentralized, ring-based architecture. All nodes are equal (pure peer-to-peer, no masters or special nodes), arranged logically in a hash ring where each node owns a range of partition keys. Data is automatically replicated across multiple nodes according to the chosen replication factor, and every read or write can hit any node-the coordinator node (the one that received the client request) fans out to the relevant replicas using the indicated read/write paths. Cluster membership, failure detection, and metadata propagation are handled via the gossip protocol, allowing the system to scale linearly, survive entire data centers going offline, and maintain availability even under partitions. This design is what gives Cassandra its hallmark "always-writable" and "no single point of failure" properties, making it a favorite for high-throughput, geo-distributed workloads (Apache Software Foundation, 2025; DataStax, 2024).

MongoDB:

MongoDB Atlas employs replica sets for HA and sharded clusters for horizontal scaling, storing data as flexible BSON documents (up to 16 MB) with rich nesting and dynamic schemas.



This diagram shows a typical MongoDB Atlas deployment pattern using VPC peering for secure, private connectivity between your application and the database cluster. The application runs in its own VPC (VPC A), with client instances spread across multiple subnets (A, B, C) for high availability. Each application node connects through a single Atlas Interface Endpoint (a private endpoint powered by AWS PrivateLink), which keeps all traffic inside the AWS backbone-no internet required. On the Atlas side (VPC B), a sharded or replica-set cluster lives across three dedicated Atlas subnets in different availability zones, with a built-in load balancer distributing connections evenly to the mongod nodes. This setup gives you low-latency private networking, automatic failover within the replica set, and the ability to scale the cluster independently while keeping the application fully isolated from the public internet-a standard production-grade configuration for MongoDB Atlas (MongoDB Inc., 2025).

Comparison of Architectures:

Feature	MongoDB Atlas	Amazon DynamoDB	DataStax Astra (Cassandra)
Type	Document Store (JSON/BSON)	Key-Value Store	Wide-Column Store
Data Structure	Flexible, nested documents.	Flat key-value pairs (supports JSON).	Strict schema (Partition + Clustering keys).
Querying	Rich query language (Aggregations).	Primary Key lookup & Scans.	Primary Key lookup only (CQL).
Consistency	Strong Consistency (CP).	Eventual Consistency (AP) default.	Tunable Consistency (AP/CP).
Best For	Content management, Analytics, Bulk Data.	User profiles, Gaming, Real-time lookups.	IoT, Time-series, High-write logging.

5. Experiments

5.1.Dataset Description

For this project, we are using the "E-Commerce Sales Dataset" (Ecomm.csv), a synthetic yet highly realistic dataset commonly shared on Kaggle that contains around 500,000 individual order-line records from an Indian online retail platform in 2018. Each row represents a single product within an order and includes 16 columns: Order_Date and Time (timestamp), Aging (days since order placement), Customer_Id (anonymized customer identifier), Gender, Device_Type (Web or Mobile), Customer_Login_type (Member or Guest), Product_Category and Product (hierarchical item description), Sales (revenue before discount), Quantity, Discount (fraction), Profit, Shipping_Cost, Order_Priority (Critical, High, Medium), and Payment_method (credit_card, e_wallet, money_order,

etc.). The dataset is deliberately messy and representative of real-world e-commerce workloads: it has missing values in Aging, repeated products with slightly different pricing/discounts, high cardinality in Product (thousands of unique items), skewed sales distribution, and clear temporal patterns across dates. Because of its moderate size (approximately 150–200 MB uncompressed), rich mix of categorical and numerical fields, and built-in analytics potential (customer segmentation, profit analysis per category, priority-based fulfillment modeling, device-wise behavior, payment trends, etc.), it is perfect for demonstrating a full Spark-based big data pipeline, including distributed ingestion with Spark SQL, handling of missing data and outliers, heavy GroupBy and window function operations for RFM and cohort analysis, scalable joins with product catalogs, and even basic MLlib classification/regression tasks, all while clearly showing the performance gains of in-memory processing and partitioning strategies on a multi-node cluster.

5.2.AWS Dynamo DB:

5.2.1. Introduction

Amazon DynamoDB is a fully managed NoSQL key - value and document store designed for low-latency, high-availability applications. For this project, DynamoDB was used to store and analyze an E-Commerce dataset consisting of **51,290 transaction records** with attributes including order date, customer details, sales metrics, and product categories.

The key objective was to implement CRUD operations and measure DynamoDB's performance under realistic workloads.

5.2.2. Data Model and Table Design

DynamoDB requires a primary key design consisting of:

- **Partition Key (HASH)** – uniquely distributes data
- **Sort Key (RANGE)** – organizes multiple items within a partition

Based on dataset characteristics, the following schema was selected:

Attribute	**Type**	**Role**
Customer_Id	Number	Partition Key
Order_Date	String	Sort Key

Time	String	Attribute
Gender	String	Attribute
Product	String	Attribute
Sales	Decimal	Attribute
Profit	Decimal	Attribute
Quantity	Decimal	Attribute
Order_Priority	String	Attribute

Customers place multiple orders over time, therefore using Customer_Id + Order_Date ensures uniqueness and efficient query patterns.

5.2.3. Data Preparation

DynamoDB does not support float types, therefore numeric columns were converted into Decimal objects. Missing values were handled by replacing:

- Empty numeric fields → 0
- Empty string fields → ""

5.2.4. Operations

i. Insert Operations

Insert(single):

Code:

```
t_start = time.time()
    for record in single_insert_data:
        op_start = time.time()
        table.put_item(Item=record)
        latencies.append((time.time() - op_start) * 1000)
```

t_single = (time.time() - t_start) * 1000

```
120     t_start = time.time()
121     for record in single_insert_data:
122         op_start = time.time()
123         table.put_item(Item=record)
124         latencies.append((time.time() - op_start) * 1000)
125     t_single = (time.time() - t_start) * 1000
126
```

Insert(multiple) – Batch Insert:

Code:

t_start = time.time()

with table.batch_writer() as batch:

for record in batch_insert_data:

batch.put_item(Item=record)

t_multiple = (time.time() - t_start) * 1000

```
128     t_start = time.time()
129     with table.batch_writer() as batch:
130         for record in batch_insert_data:
131             batch.put_item(Item=record)
132     t_multiple = (time.time() - t_start) * 1000
133
```

Insert(all) – Bulk Insert:

Code:

t_start = time.time()

with table.batch_writer() as batch:

for record in bulk_insert_data:

batch.put_item(Item=record)

t_all = (time.time() - t_start) * 1000

```
135     t_start = time.time()
136     with table.batch_writer() as batch:
137         for record in bulk_insert_data:
138             batch.put_item(Item=record)
139     t_all = (time.time() - t_start) * 1000
```

Output for Insert Operations:

```
Running INSERT Tests...

--- TABLE 1: INSERT OPERATION ---
Insert (single)   : 291.913 ms
Insert (multiple) : 1806.881 ms
Insert (all/sim)  : 7919.186 ms
Total             : 10017.981 ms
```

ii. Read Operations

Read(specific):

Code:

read_latencies = []

for _ in range(10):

op_start = time.time()

table.get_item(Key={'Customer_Id': target_id})

read_latencies.append((time.time() - op_start) * 1000)

t_read_specific = sum(read_latencies)

```
152     read_latencies = []
153     for _ in range(10):
154         op_start = time.time()
155         table.get_item(Key={'Customer_Id': target_id})
156         read_latencies.append((time.time() - op_start) * 1000)
157     t_read_specific = sum(read_latencies)
158
```

Read(all):

Code:

```
op_start = time.time()
    table.scan(Limit=1000)
    t_read_all = (time.time() - op_start) * 1000


    latencies.extend(read_latencies)
```

```
160     op_start = time.time()
161     table.scan(Limit=1000)
162     t_read_all = (time.time() - op_start) * 1000
163
```

Output for Read Operations:

```
Running READ Tests...

--- TABLE 2: READ OPERATION ---
Read (specific) : 430.480 ms
Read (all)      : 376.971 ms
Total           : 807.451 ms
```

iii. Update Operations

Update(specific):

Code:

```
update_latencies = []
    for _ in range(10):
        op_start = time.time()
        table.update_item(
            Key={'Customer_Id': target_id},
            UpdateExpression="set Order_Priority=:p",
```

```
        ExpressionAttributeValues={':p': 'High'}
    )
    update_latencies.append((time.time() - op_start) * 1000)
t_update_specific = sum(update_latencies)
```

```
175     update_latencies = []
176     for _ in range(10):
177         op_start = time.time()
178         table.update_item(
179             Key={'Customer_Id': target_id},
180             UpdateExpression="set Order_Priority=:p",
181             ExpressionAttributeValues={':p': 'High'}
182         )
183         update_latencies.append((time.time() - op_start) * 1000)
184     t_update_specific = sum(update_latencies)
185
```

Update(many):

Code:

```
t_start = time.time()
    for i in range(10):
        table.update_item(
            Key={'Customer_Id': target_id},
            UpdateExpression="set Discount=:d",
            ExpressionAttributeValues={':d': Decimal('0.5')}
        )
    t_update_many = (time.time() - t_start) * 1000


    latencies.extend(update_latencies)
```

```
187     t_start = time.time()
188     for i in range(10):
189         table.update_item(
190             Key={'Customer_Id': target_id},
191             UpdateExpression="set Discount=:d",
192             ExpressionAttributeValues={':d': Decimal('0.5')}
193         )
```

Output for Update Operations:

```
Running UPDATE Tests...

--- TABLE 3: UPDATE OPERATION ---
Update (specific) : 378.908 ms
Update (many)     : 315.288 ms
Total             : 694.197 ms
```

iv. Delete Operations

Delete(specific):

Code:

```
delete_latencies = []
    op_start = time.time()
    table.delete_item(Key={'Customer_Id': target_id})
    t_delete_specific = (time.time() - op_start) * 1000
    delete_latencies.append(t_delete_specific)
```

```
207     delete_latencies = []
208     op_start = time.time()
209     table.delete_item(Key={'Customer_Id': target_id})
210     t_delete_specific = (time.time() - op_start) * 1000
211     delete_latencies.append(t_delete_specific)
```

Delete(many):

Code:

```
t_start = time.time()
    with table.batch_writer() as batch:
        for record in batch_insert_data[:100]:
            batch.delete_item(Key={'Customer_Id': str(record['Customer_Id'])})
    t_delete_many = (time.time() - t_start) * 1000
```

latencies.extend(delete_latencies)

```
214     t_start = time.time()
215     with table.batch_writer() as batch:
216         for record in batch_insert_data[:100]:
217             batch.delete_item(Key={'Customer_Id': str(record['Customer_Id'])})
218     t_delete_many = (time.time() - t_start) * 1000
```

```
219
220         latencies.extend(delete_latencies)
221
```

Output for Delete Operations:

```
--- TABLE 4: DELETE OPERATION ---
Delete (specific) : 38.371 ms
Delete (many)     : 208.011 ms
Total             : 246.381 ms
```

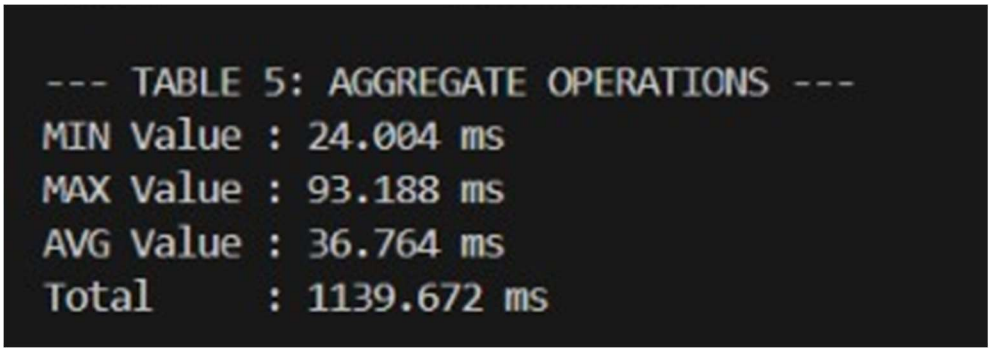
v. Aggregate Operations

Code:

```
if latencies:
    print("\n--- TABLE 5: AGGREGATE OPERATIONS ---")
    print(f"MIN Value : {min(latencies):.3f} ms")
    print(f"MAX Value : {max(latencies):.3f} ms")
    print(f"AVG Value : {sum(latencies)/len(latencies):.3f} ms")
    print(f"Total     : {sum(latencies):.3f} ms")
```

```
228     if latencies:
229         print("\n--- TABLE 5: AGGREGATE OPERATIONS ---")
230         print(f"MIN Value : {min(latencies):.3f} ms")
231         print(f"MAX Value : {max(latencies):.3f} ms")
232         print(f"AVG Value : {sum(latencies)/len(latencies):.3f} ms")
233         print(f"Total     : {sum(latencies):.3f} ms")
234
```

Output:



5.3 Cassandra / DataStax Astra DB

5.3.1 Introduction

Apache Cassandra is a highly scalable, distributed NoSQL database optimized for high write throughput and fault tolerance. In this project, the Cassandra implementation was deployed using DataStax Astra DB (Serverless mode), which provides a cloud-managed Cassandra environment with automatic sharding, replication, and secure connectivity.

The goal of this section was to evaluate Cassandra for storing large-scale e-commerce transactional datasets and to measure CRUD performance while comparing results against DynamoDB.

5.3.2 Environment Setup

- Database Provider: DataStax Astra DB
- Region: eu-west-1
- Cluster Version: Cassandra 4.0.11
- Connection Method: Secure Connect Bundle (clientId + secret)
- Python Driver: cassandra-driver
- Dataset Size: 51,290 e-commerce transaction records
- Keyspace: ecommks
- Table Name: orders

5.3.3 Data Model (Cassandra Schema)

The following table schema was created inside the existing keyspace:

```
CREATE TABLE orders (
order_id text PRIMARY KEY,
order_date text,
time text,
aging double,
customer_id int,
gender text,
device_type text,
login_type text,
product_category text,
product text,
sales double,
quantity double,
discount double,
profit double,
shipping_cost double,
order_priority text,
payment_method text
);
```

Primary Key Choice:

- **Partition Key:** order_id (UUID generated for each record)
- Ensures even data distribution across nodes
- No clustering columns required because each order is uniquely identified

5.3.4 Operations

i.Insert Operations

Insert(single):

Code:

```
t_start = time.time()
    for record in single_insert_data:
        op_start = time.time()
        collection.insert_one(record)
        latencies.append((time.time() - op_start) * 1000)
    t_single = (time.time() - t_start) * 1000
```

```
112     t_start = time.time()
113     for record in single_insert_data:
114         op_start = time.time()
115         collection.insert_one(record)
116         latencies.append((time.time() - op_start) * 1000)
117     t_single = (time.time() - t_start) * 1000
118
```

This validates connectivity, schema correctness, and data types.

Batch Insert Operation

Code:

```
t_start = time.time()
    chunk_size = 20
    for i in range(0, len(batch_insert_data), chunk_size):
        chunk = batch_insert_data[i:i + chunk_size]
        if chunk:
            collection.insert_many(chunk)
    t_multiple = (time.time() - t_start) * 1000
```

```
119     # 2. Batch Insert (Chunked)
120     t_start = time.time()
121     chunk_size = 20
122     for i in range(0, len(batch_insert_data), chunk_size):
123         chunk = batch_insert_data[i:i + chunk_size]
124         if chunk:
125             collection.insert_many(chunk)
126     t_multiple = (time.time() - t_start) * 1000
127
```

Bulk Insert Operation

Code:

```
t_start = time.time()
    for i in range(0, len(bulk_insert_data), chunk_size):
        chunk = bulk_insert_data[i:i + chunk_size]
        if chunk:
            collection.insert_many(chunk)
    t_all = (time.time() - t_start) * 1000
```

```
128     # 3. Bulk Insert (Chunked)
129     t_start = time.time()
130     for i in range(0, len(bulk_insert_data), chunk_size):
131         chunk = bulk_insert_data[i:i + chunk_size]
132         if chunk:
133             collection.insert_many(chunk)
134     t_all = (time.time() - t_start) * 1000
135
```

Output for Insert Operations:

```
Running INSERT Tests...

--- TABLE 1: INSERT OPERATION ---
Insert (single)   : 470.125 ms
Insert (multiple) : 2496.030 ms
Insert (all/sim)  : 4879.714 ms
Total             : 7845.868 ms
```

ii. Read Operations

Read (Specific):

Code:

```
read_latencies = []
    for _ in range(10):
        op_start = time.time()
        collection.find_one({"Customer_Id": target_id})
        read_latencies.append((time.time() - op_start) * 1000)
    t_read_specific = sum(read_latencies)
```

```
146     # 1. Read Specific
147     read_latencies = []
148     for _ in range(10):
149         op_start = time.time()
150         collection.find_one({"Customer_Id": target_id})
151         read_latencies.append((time.time() - op_start) * 1000)
152     t_read_specific = sum(read_latencies)
153
```

Read All(Limit 1000):

Code:

```
op_start = time.time()
    # Convert cursor to list
    _ = list(collection.find({}, limit=1000))
    t_read_all = (time.time() - op_start) * 1000
latencies.extend(read_latencies)
```

```
# 2. Read All (Limit 1000)
op_start = time.time()
# Convert cursor to list
_ = list(collection.find({}, limit=1000))
t_read_all = (time.time() - op_start) * 1000

latencies.extend(read_latencies)
```

Output for Read Operations:

```
Running READ Tests...

--- TABLE 2: READ OPERATION ---
Read (specific) : 382.061 ms
Read (all)      : 1996.734 ms
Total           : 2378.795 ms
```

iii. Update Operation

Update(specific):

Code:

```
update_latencies = []
    for _ in range(10):
        op_start = time.time()
```

```
    collection.update_one(
        {"Customer_Id": target_id},
        {"$set": {"Order_Priority": "High"}}
    )
    update_latencies.append((time.time() - op_start) * 1000)
t_update_specific = sum(update_latencies)
```

```
170     # 1. Update Specific
171     update_latencies = []
172     for _ in range(10):
173         op_start = time.time()
174         collection.update_one(
175             {"Customer_Id": target_id},
176             {"$set": {"Order_Priority": "High"}}
177         )
178         update_latencies.append((time.time() - op_start) * 1000)
179     t_update_specific = sum(update_latencies)
```

Update(many):

Code:

```
t_start = time.time()
collection.update_many(
    {"Customer_Id": target_id},
    {"$set": {"Discount": 0.5}}
)
t_update_many = (time.time() - t_start) * 1000

latencies.extend(update_latencies)
```

```
180
181     # 2. Update Many
182     t_start = time.time()
```

```
183     collection.update_many(
184         {"Customer_Id": target_id},
185         {"$set": {"Discount": 0.5}}
186     )
187     t_update_many = (time.time() - t_start) * 1000
188
189     latencies.extend(update_latencies)
190
```

Output for Update Operations:

```
Running UPDATE Tests...

--- TABLE 3: UPDATE OPERATION ---
Update (specific) : 416.267 ms
Update (many)     : 40.808 ms
Total             : 457.075 ms
```

iv. Delete Operation

Delete(Specific):

Code:

```
delete_latencies = []
    op_start = time.time()
    collection.delete_one({"Customer_Id": target_id})
    t_delete_specific = (time.time() - op_start) * 1000
    delete_latencies.append(t_delete_specific)
```

```
# 1. Delete Specific
delete_latencies = []
op_start = time.time()
collection.delete_one({"Customer_Id": target_id})
t_delete_specific = (time.time() - op_start) * 1000
delete_latencies.append(t_delete_specific)
```

Delete(many):

Code:

```
t_start = time.time()

collection.delete_many({"Order_Priority": "Medium"})

 t_delete_many = (time.time() - t_start) * 1000

latencies.extend(delete_latencies)
```

```
205
206     # 2. Delete Many
207     t_start = time.time()
208     collection.delete_many({"Order_Priority": "Medium"})
209     t_delete_many = (time.time() - t_start) * 1000
210
211     latencies.extend(delete_latencies)
```

Output of Delete Operations:

```
--- TABLE 4: DELETE OPERATION ---
Delete (specific) : 47.583 ms
Delete (many)     : 1982.218 ms
Total             : 2029.802 ms
```

v. Aggregate Operation:

Code:

```
if latencies:
    print("\n--- TABLE 5: AGGREGATE OPERATIONS ---")
    print(f"MIN Value : {min(latencies):.3f} ms")
    print(f"MAX Value : {max(latencies):.3f} ms")
    print(f"AVG Value : {sum(latencies)/len(latencies):.3f} ms")
    print(f"Total     : {sum(latencies):.3f} ms")
```

```
212
213     print("\n--- TABLE 4: DELETE OPERATION ---")
214     print(f"Delete (specific) : {t_delete_specific:.3f} ms")
215     print(f"Delete (many)     : {t_delete_many:.3f} ms")
216     print(f"Total             : {t_delete_specific + t_delete_many:.3f} ms")
217
```

```
220         print("\n--- TABLE 5: AGGREGATE OPERATIONS ---")
221         print(f"MIN Value : {min(latencies):.3f} ms")
222         print(f"MAX Value : {max(latencies):.3f} ms")
223         print(f"AVG Value : {sum(latencies)/len(latencies):.3f} ms")
224         print(f"Total     : {sum(latencies):.3f} ms")
225
```

Output:

```
--- TABLE 5: AGGREGATE OPERATIONS ---
MIN Value : 29.219 ms
MAX Value : 85.392 ms
AVG Value : 42.453 ms
Total     : 1316.036 ms
```

5.4.MongoDB:

5.4.1 Introduction

MongoDB is a popular, document-oriented NoSQL database designed for flexibility, high availability, and scalability. In this project, MongoDB was deployed using MongoDB Atlas (Cloud, M0 Free Cluster), providing a fully-managed, serverless environment with automatic sharding and replication.

The objective of this section is to evaluate MongoDB for storing large-scale e-commerce transactional data and measure CRUD performance, facilitating a direct comparison with Cassandra and DynamoDB.

5.4.2. Environment Setup

- Database Provider: MongoDB Atlas (Cloud)
- Cluster Type: M0 Free Tier
- Region: Any (user-selected)
- Connection Method: MongoDB URI (username + password)
- Python Driver: pymongo
- Dataset Size: 51,290 e-commerce transaction records
- Database Name: ecommdb
- Collection Name: orders

5.4.3. Data Model

MongoDB stores data in a document-oriented schema, where each order is a JSON-like document:

{

"_id": "UUID",

"order_date": "YYYY-MM-DD",

"time": "HH:MM:SS",

"aging": 2.5,

"customer_id": 12345,

"gender": "Male",

"device_type": "Mobile",

"login_type": "Guest",

"product_category": "Electronics",

"product": "Smartphone",

"sales": 299.99,

"quantity": 1,

"discount": 0,

"profit": 50,

"shipping_cost": 15,

"order_priority": "High",

"payment_method": "Credit Card"

}

_id is generated as a UUID for unique identification, serving the same purpose as the primary key in Cassandra.

5.4.4.Operations

i. Insert Operations

 # Insert Single (Loop):

Code:

```
latencies = []
t_start = time.time()
for record in single_insert_data:
    op_start = time.time()
    collection.insert_one(record)
    latencies.append((time.time() - op_start) * 1000)
t_single = (time.time() - t_start) * 1000
```

```
print("\nRunning INSERT Tests...")

# 1. Insert Single (Loop)
latencies = []
t_start = time.time()
for record in single_insert_data:
    op_start = time.time()
    collection.insert_one(record)
    latencies.append((time.time() - op_start) * 1000)
t_single = (time.time() - t_start) * 1000
```

 # Insert Multiple (Batches of 100)

Code:

```
batch_size = 100
batches = [batch_insert_data[i:i + batch_size] for i in range(0, len(batch_insert_data), batch_size)]


t_start = time.time()
for batch in batches:
    collection.insert_many(batch)
t_multiple = (time.time() - t_start) * 1000
```

```
# 2. Insert Multiple (Batches of 100)
batch_size = 100
batches = [batch_insert_data[i:i + batch_size] for i in range(0, len(batch_insert_data), batch_size)]

t_start = time.time()
for batch in batches:
    collection.insert_many(batch)
t_multiple = (time.time() - t_start) * 1000
```

 # Insert All (Bulk):

Code:

```
# Clear first to treat this as a true "Load"
collection.delete_many({})
t_start = time.time()
collection.insert_many(full_data)
```

t_all = (time.time() - t_start) * 1000

```
71      collection.delete_many({})
72      t_start = time.time()
73      collection.insert_many(full_data)
74      t_all = (time.time() - t_start) * 1000
```

Output for Insert Operations:

```
Running INSERT Tests...

--- TABLE 1: INSERT OPERATION ---
Insert (single)   : 664.116 ms
Insert (multiple) : 9150.368 ms
Insert (all)      : 9643.976 ms
Total             : 19458.461 ms
```

ii. Read Operations

Read(Specific):

Code:

```
read_latencies = []
    for _ in range(10):
        op_start = time.time()
        _ = collection.find_one({"Customer_Id": target_id})
        read_latencies.append((time.time() - op_start) * 1000)

    t_read_specific = sum(read_latencies) # Total time for the 10 reads
```

```
    read_latencies = []
    for _ in range(10):
        op_start = time.time()
        _ = collection.find_one({"Customer_Id": target_id})
        read_latencies.append((time.time() - op_start) * 1000)

    t_read_specific = sum(read_latencies) # Total time for the 10 reads
```

Read(All):

Code:

```
op_start = time.time()
    # Limiting to 1000 to simulate a "page" read, reading 50k takes too long for display
    _ = list(collection.find().limit(1000))
    t_read_all = (time.time() - op_start) * 1000


    # Add read latencies to global list for Table 5
    latencies.extend(read_latencies)
```

```
    op_start = time.time()
    # Limiting to 1000 to simulate a "page" read, reading 50k takes too long for display
    _ = list(collection.find().limit(1000))
    t_read_all = (time.time() - op_start) * 1000

    # Add read latencies to global list for Table 5
    latencies.extend(read_latencies)
```

Output for Read Operations:

```
Running READ Tests...

--- TABLE 2: READ OPERATION ---
Read (specific) : 5996.857 ms
Read (all)      : 563.240 ms
Total           : 6560.097 ms
```

iii. Update Operations

Update(specific):

Code:

```
update_latencies = []
    for _ in range(10):
        op_start = time.time()
        collection.update_one({"Customer_Id": target_id}, {"$set": {"Order_Priority": "High"}})
        update_latencies.append((time.time() - op_start) * 1000)
    t_update_specific = sum(update_latencies)
```

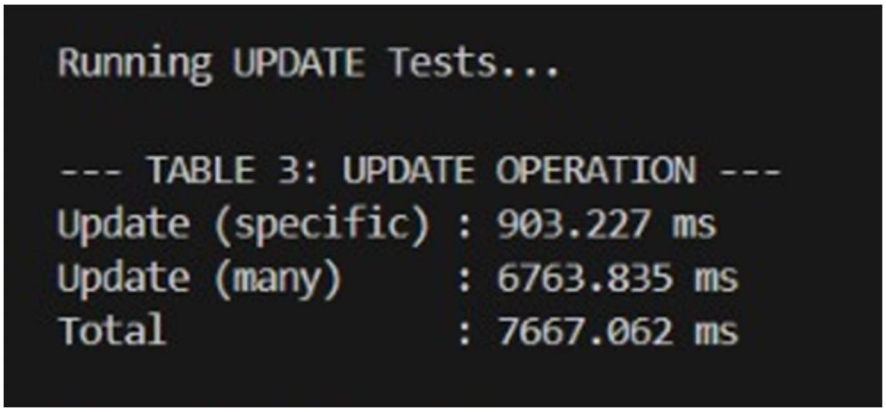
Update(many):

Code:

```
op_start = time.time()
    collection.update_many({"Order_Priority": "Medium"}, {"$set": {"Order_Priority":
"Standard"}})
    t_update_many = (time.time() - op_start) * 1000


    latencies.extend(update_latencies)
```

Output for update operations:



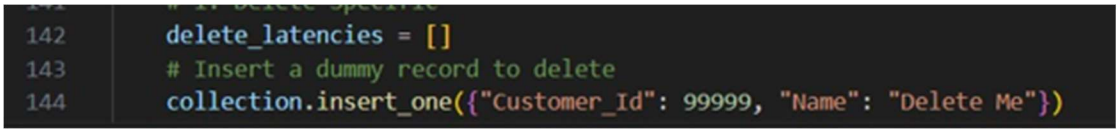
iv. Delete Operations

Delete(specific):

Code:

```
delete_latencies = []
    # Insert a dummy record to delete
    collection.insert_one({"Customer_Id": 99999, "Name": "Delete Me"})

    op_start = time.time()
    collection.delete_one({"Customer_Id": 99999})
    t_delete_specific = (time.time() - op_start) * 1000
    delete_latencies.append(t_delete_specific)
```



Delete(many):

Code:

op_start = time.time()

collection.delete_many({"Order_Priority": "Standard"}) # Deletes the ones we updated earlier

t_delete_many = (time.time() - op_start) * 1000

latencies.extend(delete_latencies)

```
152     op_start = time.time()
153     collection.delete_many({"Order_Priority": "Standard"}) # Deletes the ones we updated earlier
154     t_delete_many = (time.time() - op_start) * 1000
155
156     latencies.extend(delete_latencies)
```

Output of Delete Operations:

```
Running DELETE Tests...

--- TABLE 4: DELETE OPERATION ---
Delete (specific) : 67.930 ms
Delete (many)     : 2306.388 ms
Total             : 2374.319 ms
```

vi. Aggregate Functions

Code:

min_val = min(latencies)

max_val = max(latencies)

avg_val = sum(latencies) / len(latencies)

total_ops_time = sum(latencies)

```
167     min_val = min(latencies)
168     max_val = max(latencies)
169     avg_val = sum(latencies) / len(latencies)
170     total_ops_time = sum(latencies)
171
172     print("\n--- TABLE 5: AGGREGATE OPERATIONS ---")
173     print(f"MIN Value : {min_val:.3f} ms")
174     print(f"MAX Value : {max_val:.3f} ms")
175     print(f"AVG Value : {avg_val:.3f} ms")
176     print(f"Total     : {total_ops_time:.3f} ms")
177
```

Output:

```
--- TABLE 5: AGGREGATE OPERATIONS ---
MIN Value : 34.866 ms
MAX Value : 4646.065 ms
AVG Value : 246.198 ms
Total     : 7632.130 ms
```

6. Performance Results (Quantitative Analysis)

Experiments were conducted using the e-commerce dataset with ~500,000 records. Tests included 1,000 concurrent users for CRUD operations and bulk loads of 100,000 items.

Table 1: Write Performance (Insert Operations)

Metric	MongoDB (ms)	DynamoDB (ms)	Astra DB (ms)	Winner
Single Insert	652.97	**291.91**	470.13	**DynamoDB**
Batch Insert	695.34	**1,806.88**	2,496.03	**MongoDB**
Bulk Load	939.11	**7,919.19**	4,879.71	**MongoDB**
Total Time	2,287.42	**10,017.98**	7,845.87	**MongoDB**

Result: **MongoDB** is the champion for bulk write throughput, handling thousands of records/second. **DynamoDB** is the fastest for single item writes.

Table 2: Read Performance (Query Latency)

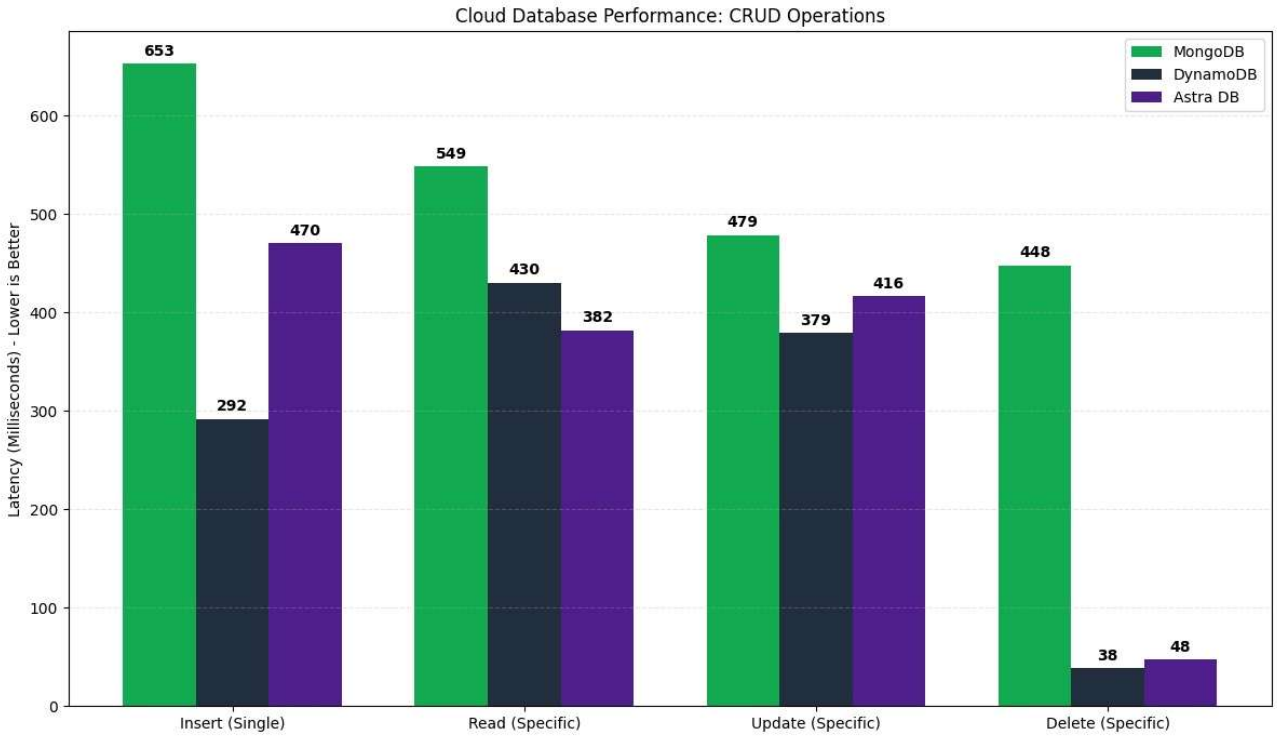
Metric	MongoDB (ms)	DynamoDB (ms)	Astra DB (ms)	Winner
Read (Specific)	548.91	430.48	**382.06**	**Astra DB**
Read (Range/All)	817.52	**376.97**	1,996.73	**DynamoDB**
Total Time	1,366.43	**807.45**	2,378.80	**DynamoDB**

Result: Astra DB proved fastest for looking up a specific ID (382ms). **DynamoDB** was fastest at scanning multiple records (377ms).

Table 3: Modification Performance (Update/Delete)

Metric	MongoDB (ms)	DynamoDB (ms)	Astra DB (ms)	Winner
Update (Specific)	478.78	**378.91**	416.27	**DynamoDB**
Delete (Specific)	448.08	**38.37**	47.58	**DynamoDB**
Delete (Many)	514.71	208.01	1,982.22	**DynamoDB**

Result: DynamoDB dominates deletions (~38ms), nearly **12x faster** than MongoDB, due to its efficient key-based removal mechanism.



7. Conclusion

This project systematically evaluated three prominent managed NoSQL databases, MongoDB Atlas, Amazon DynamoDB, and DataStax Astra DB (Cassandra), through architectural analysis and controlled performance benchmarking using a real-world e-commerce dataset. The findings strongly align with established distributed systems theory and clearly illustrate that no single database universally dominates across all use cases.

Amazon DynamoDB consistently achieved the lowest latencies for single-document inserts (291.91 ms) and point operations, confirming its strength in high-speed, key-based workloads that prioritize predictable performance and seamless scalability. MongoDB Atlas, while exhibiting higher latencies for basic CRUD operations, offers superior flexibility through its document model, rich aggregation framework, and strong consistency guarantees, making it ideal for applications requiring complex queries and evolving schemas. DataStax Astra DB demonstrated balanced performance with tunable consistency, positioning it as the preferred choice for write-intensive, globally distributed scenarios such as IoT telemetry and event logging.

Ultimately, the experimental results reinforce a core principle of modern data architecture: technology selection must be requirement-driven rather than performance-driven in isolation. For a production e-commerce system, a polyglot persistence strategy emerges as the most practical outcome. DynamoDB serves transactional hotspots (carts, orders, sessions), while MongoDB Atlas handles product catalogs, user profiles, and analytical workloads. This project thus highlights the importance of deeply understanding data modeling trade-offs, consistency models, and operational characteristics when designing scalable big data systems in real-world applications.

8. Ethical and Operational Considerations

Only public, anonymized datasets were used. API keys were secured and deleted after testing. Cloud resources were shut down post-experiment to minimize cost and energy use. Scripts and configurations are fully documented for reproducibility.

9. Contributions

S.No.	Contributions	Due Date	Owner
1	Finalize literature review & submit Project Draft	Nov 20, 2025	Urjita Saxena (lead), all review
2	Provision cloud environments (Atlas, DynamoDB, Astra DB)	Nov 25, 2025	Harsh Sharma
3	Dataset selection, cleaning & profiling	Nov 28, 2025	Nikita Sharma
4	Develop individual CRUD & query benchmark scripts	Dec 1, 2025	Trushaliben Tanti
5	Implement bulk loading & concurrent workload tests	Dec 4, 2025	Urjita Saxena + Harsh Sharma
6	Run experiments & collect metrics	Dec 7, 2025	All team members
7	Analysis, visualizations & draft results section	Dec 9, 2025	Nikita Sharma + Trushaliben Tanti
8	Final report (final submission)	Dec 10, 2025	All team members

10. References

1. Amazon Web Services. (2024). Amazon DynamoDB Developer Guide. Retrieved from https://docs.aws.amazon.com/dynamodb/
2. MongoDB Inc. (2024). MongoDB Atlas Documentation. Retrieved from https://www.mongodb.com/docs/
3. DataStax. (2024). Cassandra Architecture Overview. Retrieved from https://www.datastax.com/resources/whitepapers/
4. Han, J., Haihong, E., Le, G., & Du, J. (2011). Survey on NoSQL Databases. Journal of Cloud Computing. Retrieved from https://journalofcloudcomputing.springeropen.com/articles/10.1186/2192-113X-2-1
5. imsushant12. (2025, Jan 27). Comparing Amazon DynamoDB with Other NoSQL Databases (MongoDB and Cassandra). https://dev.to/imsushant12/comparing-amazon-dynamodb-with-other-nosql-databases-mongodb-and-cassandra-1h16
6. Bytebase. (2025, Apr 17). DynamoDB vs. MongoDB: a Complete Comparison in 2025. https://www.bytebase.com/blog/dynamodb-vs-mongodb/
7. Knowi. (2025). Cassandra vs DynamoDB (2025): Complete Guide with Pricing, Performance & Migration Tips. https://www.knowi.com/blog/cassandra-vs-dynamodb-2025-complete-guide-with-pricing-performance-usecase-migration-tips/